Automatically Partitioning Data to Facilitate the Parallelization of Functional Programs

Michael Dever(✉) and G.W. Hamilton

Dublin City University, Dublin 9, Ireland
{mdever,hamilton}@computing.dcu.ie

Abstract. In this paper we present a novel transformation technique which, given a program defined on any data-type, automatically derives conversion functions for data of that type to and from well-partitioned *join*-lists. Using these conversion functions we employ existing program transformation techniques in order to redefine the given program into an implicitly parallel one defined in terms of well-partitioned data.

1 Introduction

The development of parallel software is inherently more difficult than that of sequential software and developers can have problems thinking in a parallel setting [16]. As the limitations of single-core processor speeds are reached, the developer has no choice but to reach for parallel implementations to obtain the required performance increases.

There are many existing automated parallelization techniques [2,3,8–11,17–19], which, while powerful, require that their inputs are defined using a *cons*-list. This is an unreasonable burden to place upon a developer as it may not be intuitive to define their program in terms of a *cons*-list. In order to remove this burden, this paper presents an automatic transformation for programs which automatically partitions the data they are defined on and uses distillation [7] to redefine these programs into implicitly parallel ones defined on the resulting well-partitioned data.

The remainder of this paper is structured as follows: Sect. 2 details the language used throughout this paper. Section 3 details the transformation which converts a given program into one defined on well-partitioned data. Section 4 presents an example program, whose data is well-partitioned using our technique and an implicitly parallel program automatically derived on this well-partitioned data. Section 5 presents a summary of related work and compares our techniques with this work. Section 6 presents our conclusions and plans for further work.

2 Language

We use a standard Haskell-like higher-order functional language throughout this paper, with the usual *cons*-list notations, where data-types are defined as shown

A. Voronkov and I. Virbitskaite (Eds.): PSI 2014, LNCS 8974, pp. 59–66, 2015.
DOI: 10.1007/978-3-662-46823-4_5

$$\begin{array}{rll} t ::= & \alpha & \text{Type Variable} \\ \mid & T\ t_1 \ldots t_g & \text{Type Application} \\ & & \\ \mathbf{data}\ T\ \alpha_1 \ldots \alpha_k ::= & c_1\ t_{1_1}\ \ldots\ t_{1_{n_1}} & \text{Data-Type} \\ \vdots & & \\ \mid & c_m\ t_{m_1}\ \ldots\ t_{m_{n_m}} & \end{array}$$

Fig. 1. Data-type definition

in Fig. 1. Within this language, a data-type T can be defined with the constructors $c_1, \ldots, c_m$. Polymorphism is supported via the use of type variables, α. We use $(e :: t)$ to denote an expression e of type t.

Within this language, *join*-lists are defined as shown in Fig. 2. The language has some useful built-in functions: *split* which takes a *cons*-list and splits it in half returning a tuple containing the left and right halves and fst which takes a tuple and returns its first element. The function $removeAll_\tau$, given a sequence of types, removes all occurrences of the type τ from the given sequence.

$$\begin{array}{rl} data\ JList\ a ::= & Singleton\ a \\ \mid & Join\ (JList\ a)\ (JList\ a) \end{array}$$

Fig. 2. *join*-List data-type definition

3 Automatically Partitioning Data

There are many parallelization techniques which make use of well-partitioned data [1,2,8–11,13,17,18]. These can be restrictive and often require that input programs are defined on data that can be easily well-partitioned, however this may not always be intuitive. To solve this, we define a transformation technique that allows for the automatic partitioning of any data. An overview of this technique is shown in Fig. 3. The technique is combined with distillation in order to automatically convert a program into one defined on well-partitioned data. The technique consists of four steps:

1. Given a program defined on a data-type, τ, define a corresponding data-type, τ', instances of which contain the non-inductive components from data of type τ.
2. Derive a *partitioning function*, $partition_\tau$, which allows data of type τ to be converted into a well-partitioned *join*-list containing data of type τ'.
3. Derive a *rebuilding function*, $rebuild_\tau$, which converts a *join*-list containing data of type τ' into data of type τ.
4. Distill a program equivalent to the given program which is defined on a well-partitioned *join*-list.

Using these four steps, we can automatically convert a given program into an equivalent program defined on well-partitioned data.

$$\begin{tikzcd}
\boxed{x :: \tau} \arrow[r, bend left, "partition_\tau"] & \boxed{x' :: JList\ \tau'} \arrow[l, bend left, "rebuild_\tau"]
\end{tikzcd}$$

Fig. 3. Data partitioning functions

$$\mathcal{N}_\tau = \begin{cases} JList\ \tau' & \textbf{if } \tau \in \gamma \\ \tau' & \textbf{otherwise} \end{cases}$$

where

τ is defined by:

$$\begin{aligned} \textbf{data}\ T\ T_1 \ldots T_k ::= {} & c_1\ t_{1_1}\ \ldots\ t_{1_{n_1}} \\ & \vdots \\ \mid {} & c_m\ t_{m_1}\ \ldots\ t_{m_{n_m}} \end{aligned}$$

τ' is a new data-type defined by:

$$\begin{aligned} \textbf{data}\ T' ::= {} & c_1'\ \mathcal{N}_{t_{1_{j_1}}}\ \ldots\ \mathcal{N}_{t_{1_{k_1}}} \\ & \vdots \\ \mid {} & c_m'\ \mathcal{N}_{t_{m_{j_m}}}\ \ldots\ \mathcal{N}_{t_{m_{k_m}}} \end{aligned}$$

$$\textbf{where } \langle t_{i_{j_i}}, \ldots, t_{i_{k_i}} \rangle = \begin{cases} removeAll_\tau\ \langle t_{i_{1_i}}, \ldots, t_{i_{n_i}} \rangle & \textbf{if } \tau \in \gamma \\ \langle t_{i_{1_i}}, \ldots, t_{i_{n_i}} \rangle & \textbf{otherwise} \end{cases}$$

Fig. 4. Transformation rule for defining τ' using τ

3.1 Defining Partitioned Data-Types

To partition data of a given instantiated data-type, $\tau = T\ T_1 \ldots T_k$, we first define a corresponding data-type, τ', according to the rules shown in Fig. 4. In some cases it may not make sense to parallelize the processing of all data in a given program. To allow for this, we allow the developer to specify a set of *parallelizable-types*, γ, instances of which will be evaluated in parallel.

Given a program defined on a data-type, τ, $\mathcal{N}$ is applied to τ as follows:

- If τ is a parallelizable-type, its data must be well-partitioned and is therefore replaced by $JList\ \tau'$.
- If τ is not a parallelizable-type, it is replaced by τ' as it may contain data that must be well-partitioned.

When replacing τ with either τ' or $JList\ \tau'$, τ' is defined as shown in Fig. 4. For each constructor, c, in the definition of τ, a new constructor, c', is added to τ'. If τ is a parallelizable-type then any inductive components c contains are not added to c'. $\mathcal{N}$ is applied to each component of c'.

3.2 Converting Data to and From *Join*-Lists

To partition data of a given type, τ, we define a partitioning function, $partition_\tau$ as follows:

- If τ is a parallelizable-type, data of type τ is converted to a *cons*-list containing data of type τ', which is then well-partitioned into a *join*-list.
- If τ is not a parallelizable-type, data of type τ is converted to data of type τ', the components of which are also well-partitioned where necessary.

In order to convert a well-partitioned *join*-list back into its original form, we also define a rebuilding function, $rebuild_\tau$, the definition of which is simply the inverse of $partition_\tau$.

3.3 Distilling Programs on Well-Partitioned Data

Distillation [7] is a powerful fold/unfold based program transformation technique which eliminates intermediate data-structures from higher-order functional programs. It is capable of obtaining a super-linear increase in efficiency.

Given a sequential program, f, defined on a type, τ, we first define $partition_\tau$ and $rebuild_\tau$. Once these have been defined, we can convert f into an equivalent program, f_{wp}, defined on well-partitioned data. Applying distillation to $f \circ rebuild_\tau$, results in the automatic derivation of f_{wp}. A high level overview of this process is presented in Fig. 5.

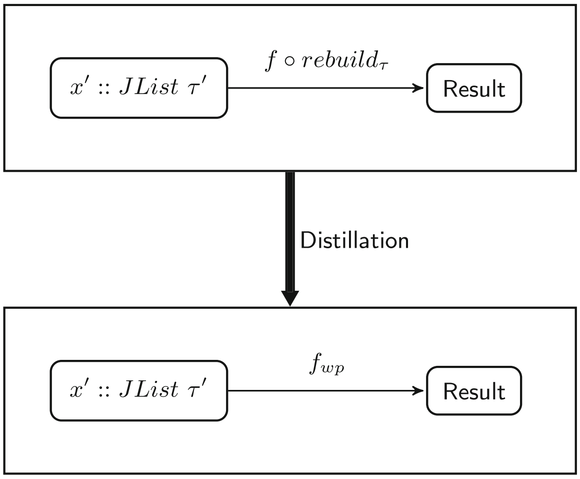

Fig. 5. Distillation of programs on well-partitioned data

As f_{wp} is defined on a well-partitioned *join*-list and f is defined on data of type τ, we must also generate the correct input for f_{wp} by applying $partition_\tau$ to the input of f.

4 Example Parallelization Using Well-Partitioned Data

Consider a function, *sumList*, which calculates the sum of a *List* of numbers:

$$\begin{array}{l} sumList = \lambda xs.\mathbf{case}\ xs\ \mathbf{of} \\ \qquad \begin{array}{ll} Nil & \rightarrow 0 \\ Cons\ x\ xs & \rightarrow x + sumList\ xs \end{array} \end{array}$$

As the input to *sumList* is of type (*List Int*), the first step in the parallelization process is to define *List*′ according to the rules shown in Fig. 4, where $\gamma = \{List\ Int\}$, as shown below:

$$\begin{array}{rl} data\ List' ::= & Nil' \\ | & Cons'\ Int \end{array}$$

The second step is to define both $partition_{(List\ Int)}$ and $rebuild_{(List\ Int)}$ as follows:

$$\begin{array}{ll} partition_{(List\ Int)} & = partition \circ flatten_{(List\ Int)} \\ \\ flatten_{(List\ Int)} & = \lambda xs.\mathbf{case}\ xs\ \mathbf{of} \\ & \qquad \begin{array}{ll} Nil & \rightarrow [Nil'] \\ Cons\ x_1\ x_2 & \rightarrow [Cons'\ x_1] +\!\!\!+ flatten_{(List\ Int)}\ x_2 \end{array} \\ \\ rebuild_{(List\ Int)} & = fst \circ unflatten_{(List\ Int)} \circ rebuild \\ \\ unflatten_{(List\ Int)} & = \lambda xs.\mathbf{case}\ xs\ \mathbf{of} \\ & \qquad (x : xs) \rightarrow \mathbf{case}\ x\ \mathbf{of} \\ & \qquad\qquad \begin{array}{ll} Nil' & \rightarrow (Nil,\ xs) \\ Cons'\ x_1 & \rightarrow \mathbf{case}\ unflatten_{(List\ Int)}\ xs\ \mathbf{of} \\ & \qquad (x_2,\ xs_2) \rightarrow (Cons\ x_1\ x_2, xs_2) \end{array} \end{array}$$

Following the definition of $rebuild_{(List\ Int)}$, we compose this with the original function, $sumList \circ rebuild_{(List\ Int)}$, and apply distillation to this composition. This allows *sumList* to be automatically redefined into an equivalent program, $sumList_{wp}$, defined in terms of a *join*-list containing instances of *List*′, resulting in the following implicitly parallel definition:

$$\begin{array}{l} sumList_{wp} = \lambda x.\mathbf{case}\ x\ \mathbf{of} \\ \qquad \begin{array}{lll} Singleton\ x & \rightarrow & \mathbf{case}\ x\ \mathbf{of} \\ & & \quad Nil' \rightarrow 0 \\ Join\ l\ r & \rightarrow & \mathbf{let}\ l' = sumList'_{wp}\ l \\ & & \qquad r' = sumList_{wp}\ r \\ & & \mathbf{in}\ \ l'\ r' \end{array} \end{array}$$

```
sumList'_wp = λx n.case x of
                     Singleton x → case x of
                                     Nil'     → n
                                     Cons' x → x + n
                     Join l r     → let l' = sumList'_wp l
                                        r' = sumList'_wp r n
                                    in  l' r'
```

By making distillation aware of the definition of the + operator, it can derive the necessary associativity that allows for each child of a *Join* to be evaluated in parallel. It is worth noting that in the case of the above program, when evaluating the left child of a *Join* we create a partial application which can be evaluated in parallel with the evaluation of the right child. This partial application is equivalent to $(\lambda r.l + r)$, where r is the result of the evaluation of the right operand. In a parallel environment the full evaluation of this partial application must be forced to ensure that the left operand has been evaluated and that parallel processes have roughly the same amount of work.

As both children are roughly equal in size, each parallel process created will have a roughly equal amount of work to do. In contrast, with respect to the original *sumList* defined on *cons*-lists, if the processing of both x and *sumList xs* are performed in parallel, one process will have one element of the list to evaluate, while the other will have the remainder of the list to evaluate, which is undesirable.

5 Related Work

There are many existing works that aim to resolve the same problem that our transformation does: mapping potentially poorly-partitioned data into a form that can be efficiently parallelized. Some work, such as list-homomorphisms and their derivative works [1,8–11,17,18] simply assume that they will use data that is well-partitioned. These techniques require that their inputs are defined using a *cons*-list, which can then be easily well-partitioned [3,5,6,19]. Restricting developers to implement their programs in these forms is an unrealistic burden.

Chin et al.'s [2] work on parallelization via context-preservation also makes use of *join*-lists as part of its parallelization process. This technique is only applicable to programs defined in the form of *list-paramorphisms* [14]. While this allows for quite a broad class of program to be parallelized, it is not realistic to force developers to define their functions in this form.

An important limitation of these techniques is that they are only applicable to lists, excluding the large class of programs that are defined on trees. One approach to parallelizing trees is that of Morihata et al.'s [15] generalization of the third homomorphism theorem [4] to trees. This approach makes use of *zippers* [12] to model the path from the root of a tree to an arbitrary leaf. While this is an interesting approach to partitioning the data contained within a binary-tree, the partitioning technique is quite complicated. It also presents no concrete methodology for generating zippers from binary-trees and assumes that the developer has provided such a function.

6 Conclusion

In this paper, we have presented a novel data-type transformation which allows for a given program to be automatically redefined into one defined on well-partitioned data. Our research is focused on automatically converting programs defined on any data-type into equivalent parallel versions defined on well-partitioned data in the form of *join*-lists. The presented data-type transformation is a significant component of that automatic parallelization system.

At a high level, by combining the outputs of the presented data-type transformation with an explicit parallelization transformation which parallelizes expressions operating on *join*-lists it is possible to automatically redefine a given sequential program defined on any data-type into an explicitly parallel one defined on well-partitioned *join*-lists.

While the presented data-type transformation is defined using *join*-lists, which appear to be the standard partitionable data-type used in automated parallelization systems [1,8–11,17,18], it is possible that *join*-lists are not the ideal data-structures to be used as part of such systems. As the data contained in a *join*-list is placed only at the leaves, it is possible that parallel processes evaluating the nodes of a *join*-list will spend much of their time waiting on the results of the parallel processes evaluating their subtrees. Further research is required to determine if there is a data structure which provides better parallel performance in general.

Where existing automated parallelization techniques are restrictive with respect to the form of their input programs and the types they are defined on, a parallelization technique defined using the presented data-type transformation should hold no such restrictions. To the best of the authors knowledge this is the first automatic data-type transformation system that will derive a well-partitioned representation of any given data-type and will redefine a given program into one defined in terms of such well-partitioned data.

Acknowledgements. This work was supported, in part, by Science Foundation Ireland grant 10/CE2/I303_1 to Lero - the Irish Software Engineering Research Centre.

References

1. Blelloch, G.E.: Scans as primitive operations. IEEE Trans. Comput. **38**(11), 1526–1538 (1989)
2. Chin, W.-N., Khoo, S.-C., Hu, Z., Takeichi, M.: Deriving parallel codes via invariants. In: Palsberg, J. (ed.) SAS 2000. LNCS, vol. 1824, pp. 75–94. Springer, Heidelberg (2000)
3. Chin, W.N., Takano, A., Hu, Z., Chin, W., Takano, A., Hu, Z.: Parallelization via context preservation. In: IEEE International Conference on Computer Languages, IEEE CS Press, pp. 153–162 (1998)

4. Gibbons, J.: The third homomorphism theorem. J. Funct. Program. **6**(4), 657–665 (1996). Earlier version appeared in Jay, C.B., (ed.), Computing: The Australian Theory Seminar, Sydney, pp. 62–69, December 1994
5. Gorlatch, S.: Systematic efficient parallelization of scan and other list homomorphisms. In: Fraigniaud, P., Mignotte, A., Robert, Y., Bougé, L. (eds.) Euro-Par 1996. LNCS, vol. 1124, pp. 401–408. Springer, Heidelberg (1996)
6. Gorlatch, S.: Systematic extraction and implementation of divide-and-conquer parallelism. In: Kuchen, H., Swierstra, S.D. (eds.) PLILP 1996. LNCS, vol. 1140, pp. 274–288. Springer, Heidelberg (1996)
7. Hamilton, G., Jones, N.: Distillation and labelled transition systems. In: Proceedings of the ACM Workshop on Partial Evaluation and Program Manipulation, pp. 15–24, January 2012
8. Hu, Z., Iwasaki, H., Takechi, M.: Formal derivation of efficient parallel programs by construction of list homomorphisms. ACM Trans. Program. Lang. Syst. **19**(3), 444–461 (1997)
9. Hu, Z., Takeichi, M., Chin, W.-N.: Parallelization in calculational forms. In: Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, POPL 1998, pp. 316–328, ACM, New York (1998)
10. Hu, Z., Takeichi, M., Iwasaki, H.: Diffusion: calculating efficient parallel programs. In: 1999 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation, pp. 85–94(1999)
11. Hu, Z., Yokoyama, T., Takeichi, M.: Program optimizations and transformations in calculation form. In: Lämmel, R., Saraiva, J., Visser, J. (eds.) GTTSE 2005. LNCS, vol. 4143, pp. 144–168. Springer, Heidelberg (2006)
12. Huet, G.: The zipper. J. Funct. Program. **7**(5), 549–554 (1997)
13. Iwasaki, H., Hu, Z.: A new parallel skeleton for general accumulative computations. Int. J. Parallel Prog. **32**, 389–414 (2004)
14. Meertens, L.: Paramorphisms. Formal Aspects Comput. **4**(5), 413–424 (1992)
15. Morihata, A., Matsuzaki, K., Hu, Z., Takeichi, M.: The third homomorphism theorem on trees: downward & upward lead to divide-and-conquer. In: Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of programming languages, POPL 1909, pp. 177–185. ACM, New York (2009)
16. Skillicorn, D.: Foundations of Parallel Programming. Cambridge International Series on Parallel Computation. Cambridge University Press, Cambridge (2005)
17. Skillicorn, D.B.: Architecture-independent parallel computation. Computer **23**, 38–50 (1990)
18. Skillicorn, D.B.: The bird-meertens formalism as a parallel model. In: Kowalik, J.S., Grandinetti, L. (eds.) Software for Parallel Computation. NATO ASI Series F, vol. 106, pp. 120–133. Springer, Heidelberg (1993)
19. Teo, Y.M., Chin, W.-N., Tan, S.H.: Deriving efficient parallel programs for complex recurrences. In: Proceedings of the Second International Symposium on Parallel symbolic computation, PASCO 1997, pp. 101–110. ACM, New York (1997)